Empir Software Eng
DOI 10.1007/s10664-017-9522-4

Identifying self-admitted technical debt in open source projects using text mining

Qiao Huang[1] · Emad Shihab[2] · Xin Xia[1,3] ·
David Lo[4] · Shanping Li[1]

Abstract Technical debt is a metaphor to describe the situation in which long-term code quality is traded for short-term goals in software projects. Recently, the concept of self-admitted technical debt (SATD) was proposed, which considers debt that is intentionally introduced, e.g., in the form of quick or temporary fixes. Prior work on SATD has shown that source code comments can be used to successfully detect SATD, however, most current state-of-the-art classification approaches of SATD rely on manual inspection of the source code comments. In this paper, we proposed an automated approach to detect SATD in source

Communicated by: Andrian Marcus

✉ Xin Xia
xxia@zju.edu.cn; xxia02@cs.ubc.ca

Qiao Huang
tkdsheep@zju.edu.cn

Emad Shihab
eshihab@encs.concordia.ca

David Lo
davidlo@smu.edu.sg

Shanping Li
shan@zju.edu.cn

[1] College of Computer Science and Technology, Zhejiang University, Hangzhou, China

[2] Data-driven Analysis of Software (DAS) Lab at the Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada

[3] Department of Computer Science, University of British Columbia, Vancouver, Canada

[4] School of Information Systems, Singapore Management University, Singapore, Singapore

Published online: 09 May 2017

code comments using text mining. In our approach, we utilize feature selection to select useful features for classifier training, and we combine multiple classifiers from different source projects to build a composite classifier that identifies SATD comments in a target project. We investigate the performance of our approach on 8 open source projects that contain 212,413 comments. Our experimental results show that, on every target project, our approach outperforms the state-of-the-art and the baselines approaches in terms of F1-score. The F1-score achieved by our approach ranges between 0.518 - 0.841, with an average of 0.737, which improves over the state-of-the-art approach proposed by Potdar and Shihab by 499.19%. When compared with the text mining-based baseline approaches, our approach significantly improves the average F1-score by at least 58.49%. When compared with a natural language processing-based baseline, our approach also significantly improves its F1-score by 27.95%. Our proposed approach can be used by project personnel to effectively identify SATD with minimal manual effort.

Keywords Technical debt · Source code comments · Text mining

1 Introduction

For most software projects, the ultimate goal is to deliver software quickly, with high quality, and on budget. However, in real-world software projects, due to various reasons (e.g., tight schedules, limited human resources, cost reduction, etc), sometimes developers have to make tradeoffs to complete their tasks in a rush.

Technical debt is a metaphor introduced by Cunningham (1993) to describe the situation where long-term code quality is traded for short-term goals. Like financial debt, technical debt incurs interest payments in the form of increased future costs owing to earlier quick and dirty design and implementation choices (Brown et al. 2010). To pay the interest or discharge of the debt, developers may need re-architecting and refactoring. Prior work has shown that technical debt is common, unavoidable and may degrade quality and increase complexity in the future (Lim et al. 2012; Wehaibi et al. 2016).

However, technical debt is not always visible. In many cases, it is (or was) only known to some people (e.g., only Bob may know that he hardcoded a parameter to quickly implement a feature before the deadline) but not visible enough to others who eventually have to pay for it (e.g., a new developer took over Bob's work and now needs to deal with the hardcoded parameter).

To solve this problem, a number of studies empirically examined technical debt and proposed different techniques to enable its detection and management. The majority of the prior work focused on using static source code analysis to detect technical debt (i.e., code smells) (Marinescu 2004; Marinescu et al. 2010). However, more recently, Potdar and Shihab (2014) proposed the concept of self-admitted technical debt (SATD), which considers debt that is intentionally introduced (e.g., quick or temporary fixes). In particular, SATD refers to the situation where developers know that the current implementation is sub-optimal and document this using code comments. For example, by manual inspection of the open source project "JEdit", we found a comment saying "Need some format checking here", which indicates that the corresponding code is defective and needs to catch format exception. A previous study (Wehaibi et al. 2016) also shows that although the percentage of SATD in a project is not high, it can have a negative impact on software complexity. Thus, we believe that it is feasible to identify certain types of technical debt through source code comments.

There are certain advantages and disadvantages of SATD. The main advantages of SATD is that it is often documented by developers who know the code, which is more reliable compared to using code metrics or code smells to detect technical debt. However, since SATD is written in plain text, it is unstructured in nature, which makes it difficult to detect. Most of the studies that focused on SATD thus far have resorted to manually classifying comments (e.g., Potdar and Shihab 2014) or using the 62 manually derived comment patterns[1] (e.g.,Wehaibi et al. 2016; Bavota and Russo 2016). Although the manually extracted comments might be useful, they have a major drawback, which is the amount of manual effort required to derive them.

Therefore, in this paper we propose a text mining-based approach to automatically detect SATD in source code comments. We build our classification model using comments from a number of different source projects as training data to predict the label of a comment in a completely new target project. In our approach, we first preprocess the text description of all comments and apply feature selection, namely Information Gain (Sebastiani 2002), on each source project to select the top 10% of features with the highest information gain scores. Then we use the selected features to train a sub-classifier on each source project. After that, a composite classifier is built from these sub-classifiers and predicts the label of a new comment in the new target project. The output of the composite classifier is the label predicted most often by its sub-classifiers, with ties broken arbitrarily.

Most recently, Maldonado et al. (2017) proposed an approach based on natural language processing (NLP) to automatically identify different types of SATD comments. In their work, they built a maximum entropy classifier based on NLP to identify the most two common types of SATD (i.e., SATD on design, SATD on requirement or non-SATD). The major difference between their work and our work is that they only focus on certain types of SATD, while we care more about whether a comment contains SATD, which also includes other types of SATD (i.e., defect debt, documentation debt and test debt). However, Maldonado et al.'s NLP-based approach could also be adapted to work in our problem. Thus, we apply their approach as a baseline in our experiment and compare its performance with our approach.

To evaluate the performance of our approach, we used a manually classified dataset of source code comments from 8 open source projects with 212,413 comments, provided by the authors of Maldonado and Shihab (2015). The experimental results show that, on every target project, our approach achieves the best performance in terms of F1-score. The F1-score achieved by our approach ranges between 0.518 - 0.841, with an average of 0.737, which is a significant improvement over the baseline approaches. On average, our approach improves the F1-scores over Potdar and Shihab's approach, Naive Bayes Multinomial (NBM) baseline, Support Vector Machine (SVM) baseline, k-Nearest Neighbor (kNN) baseline and Maldonado et al.'s NLP-based approach by 499.19%, 58.49%, 882.67% 205.81% and 27.95% respectively.

The main contributions of this paper are:

1. We propose a text mining-based approach to automatically detect SATD comment. In our approach, we utilize feature selection to select useful features for classifier training, and we combine sub-classifiers from different source projects to build a composite classifier which is more accurate in prediction.

[1] derived after the manual examination of more than 100k comments

2. We compare our approach with Potdar and Shihab's approach, several text mining-based baseline approaches and NLP-based approach on 8 open source projects with 212,413 comments. The experiment results show that our approach achieves significant improvements over these approaches.

The remainder of the paper is organized as follows. We present some background information of the current state-of-the-art classification approach of SATD comments in Section 2. We describe the overall framework and technical details of our approach in Section 3. We present our experimental setup and results in Sections 4 and 5. We discuss the implications of our study and threats to validity in Section 6. We present related work in Section 7. We conclude and mention future work in Section 8.

2 Background

In this section, we introduce the current state-of-the-art classification approach of SATD comments, which is proposed by Potdar and Shihab (2014). In their work, they used source code comments in four open source software projects - Eclipse, Chromium OS, Apache HTTP Server, and ArgoUML to identify self-admitted technical debt. They first manually read through 101,762 comments to identify those that indicated self-admitted technical debt. Then, they further distilled these comments to specific patterns that indicate self-admitted technical debt. Here a pattern represents a keyword or phrase that appears frequently in SATD comments. In total, they ended up with a set of 62 recurring patterns (notice that there are actually 63 patterns in their published data) that were identified across the four projects. A comment would be identified as SATD comment if and only if any one of these patterns appears in the text content of the comment. To make our paper self-contained, we list all these patterns in Table 1.

Based on these patterns, they performed an exploratory study on SATD, such as measuring how much SATD exists and what's the most recurring patterns across all of the projects. Several follow-on works (e.g., Wehaibi et al. (2016) and Bavota and Russo (2016)) that focused on SATD also applied these patterns to identify SATD comments.

One drawback of Potdar and Shihab's comment pattern-based approach is that it requires manual effort to summarize the patterns, which is time-consuming. Besides, our experiment results also show that there are still a large number of SATD comments not containing any one of these patterns. Thus, in this paper, we propose an automatic approach to identify SATD comments. Compared with Potdar and Shihab's comment pattern-based approach, our approach can identify more SATD comments and does not require manual summarization of patterns.

3 Approach

In this section, we first present the overall framework of our approach. Then we elaborate on the technical details of our approach, including text preprocessing, feature selection, training sub-classifiers and classifier voting.

3.1 Overall Framework

Figure 1 presents the overall framework of our approach. It contains two phases: a model building phase and a prediction phase. In the model building phase, our approach builds a

Table 1 Patterns summarized by Potdar and Shihab

Patterns that indicate SATD	
hack	ugly
nuke	barf
yuck	crap
hacky	silly
fixme	stupid
kludge	kaboom
give up	toss it
retarded	bail out
at a loss	take care
this is bs	causes issue
prolly a bug	this is wrong
fix this crap	inconsistency
is problematic	don't use this
probably a bug	this is uncool
trial and error	cause for issue
get rid of this	just abandon it
certainly buggy	remove this code
temporary crutch	some fatal error
abandon all hope	may cause problem
workaround for bug	temporary solution
this can be a mess	there is a problem
give up and go away	it doesn't work yet
this isn't very solid	something's gone wrong
this isn't quite right	is this next line safe
something bad happened	you can be unhappy now
hang our heads in shame	this doesn't look right
risk of this blowing up	is this line really safe
something bad is going on	hope everything will work
treat this as a soft error	something serious is wrong
doubt that this would work	remove me before production
this is temporary and will go away	unknown why we ever experience this
this indicates a more fundamental problem	

sub-classifier for each individual source project, using data from the other projects, with comments that have known labels (with or without SATD). In the prediction phase, we composite all sub-classifiers to jointly predict whether an unknown comment is a SATD comment or not in the target project. Note that to make our approach as practical as possible, each time we choose 1 project as "target project" (i.e., for prediction), then we denote the other *n*-1 projects as "source projects" and train our classification model using the comments from these source projects.

Our framework takes as input training comments with known labels from different source projects. It first preprocesses the text descriptions of comments and extracts features

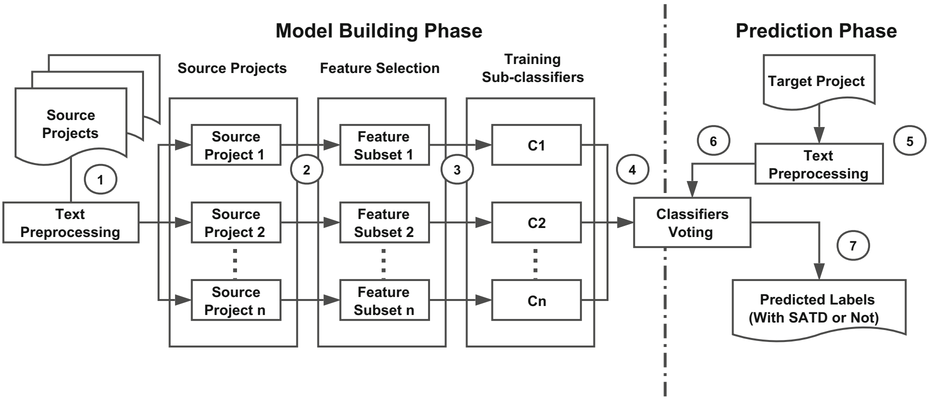


Fig. 1 Overall framework of our approach

(i.e., words) to represent each comment (Step 1). Then, for each source project, we apply feature selection to select features that are useful for classification and remove useless features (Step 2). Next, we use the selected features to train a sub-classifier for each source project (Step 3). In total, we end up with *n* classifiers which are combined to form a composite classifier for prediction (Step 4). For each new comment in the target project, we first preprocess the comment to extract features (Step 5) and input them to the composite classifier (Step 6). Finally, each sub-classifier will predict the label of the comment according to its features, and the label with the highest number of "votes" will be chosen as the final prediction result of the composite classifier (Step 7).

3.2 Text Preprocessing

We preprocess the natural-description of comments to extract features (i.e., words) in 3 steps: tokenization, stop-word removal, and stemming.

1. **Tokenization:** is the process that breaks a stream of text up into words, phrases, symbols, or other meaningful elements called tokens. In our experiment, we only keep tokens that contain English letters. Still there are some tokens attached with punctuation or numbers, we remove all these characters and only keep English letters in a token (e.g., ”TODO:” is transformed to ”TODO”). Finally, we convert all words to lowercase.
2. **Stop-word Removal:** Stop words are words that are used often and carry little meaning to distinguish different categories of comments. Examples of stop word include "I", "should", "to", "the". Although there are many text mining tookits providing a list of standard stop-words, we notice that in the domain of SATD comments, some stop words are actually useful for classification. For example, in the SATD comment "TODO: should have an image of a wizard or some logo", the phrase "should have" carries useful information, while both the words "should" and "have" are considered as stop-words by default. Thus, we manually build a list of stop-words, which only contains a small number of prepositions (e.g., "the", "to", "of", "is", etc). Words that have length no more than 2 or no less than 20 are also treated as stop-words.
3. **Stemming:** is the process of reducing inflected (or sometimes derived) words to their word stem, base or root form. For example, the words "stems", "stemmer",

and "stemmed" would all be reduced to "stem". We employ the well-known Porter stemmer[2] to reduce a word to its representative root form.

3.3 Feature Selection

After preprocessing and tokenizing the comments, we use the Vector Space Model (VSM) (Salton et al. 1975) to represent each comment with a word vector. In this model, a feature can be viewed as a dimension, and a comment can then be viewed as a data point in a high-dimensional space. In total, we have a large number of features for each source project (e.g., there are 3,661 features in ArgoUML project). An overly high number of dimensions can cause the *curse-of-dimensionality* problem (Han et al. 2006). Aside from this, we notice that only a small number of comments are SATD comments, i.e., the class imbalance problem exists (He and Garcia 2009).

To address the above problems, we apply feature selection to identify a subset of features that are most useful in differentiating different classes (i.e., comments with or without SATD). Previous studies have shown that feature selection could improve the performance of classification (Hall 1999). In this paper, we employ the widely used feature selection technique, namely Information Gain (IG) to select useful features (Sebastiani 2002; Yang and Pedersen 1997).

Let us denote a dataset of comments as $C = \{(C_1, L_1), (C_2, L_2), ..., (C_N, L_N)\}$, where C_i represents the i^{th} comment and L_i is a label that represents whether this comment is with SATD (t) or not ($\bar{t}$), and the word vector of C_i is denoted as $C_i = \{w_1, w_2, ..., w_n\}$, where n represents the number of different words appeared in C_i and w_i represents the i^{th} word. For a feature (i.e., word) w and a comment C_i, there would be 4 possible relationships:

1. (w, t): comment C_i contains the feature w, and it is a SATD comment (i.e., t).
2. $(w, \bar{t})$: comment C_i contains the feature w, but it is not a SATD comment (i.e., $\bar{t}$).
3. $(\bar{w}, t)$: comment C_i does not contains the feature w, but it is a SATD comment (i.e., t).
4. $(\bar{w}, \bar{t})$: comment C_i does not contains the feature w, and it is not a SATD comment (i.e., $\bar{t}$).

Based on the above 4 possible relationships, The information gain (IG) score of feature w and label t is computed as:

$$IG\left(w,t\right) = \sum_{t' \in \{t,\bar{t}\}} \sum_{w' \in \{w,\bar{w}\}} p\left(w',t'\right) * log \frac{p\left(w',t'\right)}{p\left(w'\right) * p\left(t'\right)} \tag{1}$$

Here *p(w',t')* represents the probability of feature *w'* appearing in a comment with label *t'*. *p(w')* represents the probability of feature *w'* appearing in a comment and *p(t')* represents the probability of a comment being with label *t'*.

Information gain (IG) measures the number of bits of information required for predicting a label (i.e., with or without SATD) by knowing the presence or absence of a feature in a comment. After we apply feature selection (i.e., IG) to compute the scores for each feature, we rank these scores from high to low to generate a ranked list. The higher the score is, the more important the feature is to distinguish a label. We select the top *k%* features whose feature selection scores are in the top *k%* of the ranked list, and remove the other features. In this way, we reduce the number of features in the model building phase, and also in the

[2] http://tartarus.org/martin/PorterStemmer/

prediction phase. By default, we empirically choose the top 10% of the total number of features. We examine the impact of using different percentages of features in our classification in Section 5.

3.4 Training Sub-classifiers

By default, we train each sub-classifier using the Naive Bayes Multinomial (NBM) technique, which is widely used in text mining (McCallum et al. 1998). Before introducing NBM, we first introduce Naive Bayes (NB) (McCallum et al. 1998). The major advantage of NB classification is its short computational training time, since it assumes that given a label (i.e., with or without SATD), features (i.e., words) are conditionally independent. Thus, given a comment $C_i = (w_1, w_2, ..., w_n)$, and a label L_i, we have:

$$p(C_i|L_i) = \prod_{j=1}^{|n|} p(w_j|L_i) \tag{2}$$

By applying Bayes Theorem on (2), we have:

$$p(L_i = t|C_i) = \frac{p(L_i = t) \times \prod_{j=1}^{|n|} p(w_j|L_i = t)}{p(C_i)} \tag{3}$$

$$p(C_i) = \sum_{t' \in \{t, \bar{t}\}} p(L_i = t') \times \prod_{j=1}^{|n|} p(w_j|L_i = t') \tag{4}$$

We can use (3) to identify the label for a comment. Notice that in NB, we only consider the presence or absence of a feature in a comment. NBM is similar to NB, but the label is determined by the number of times each feature appears in the comment. In general, when the total number of unique features in the comment collection is large, NBM may perform better than NB (Xia et al. 2014).

3.5 Classifiers Voting

In the prediction phase, we need to use the classifiers trained on source projects to predict the label of a new comment in the target project. Since each project has domain-specific comments and different feature distributions, the sub-classifier trained on a single source project may not capture the important features in the target project. So in our approach, a composite classifier is built from these sub-classifiers and it will predict the label of a new comment in the target project. The output of the composite classifier is the label predicted most often by its sub-classifiers, with ties broken arbitrarily. Thus, the prediction phase is just like an election, each sub-classifier will vote to decide the final winner (i.e., the label of a new comment). A similar idea of the voting strategy can be found in the well-known ensemble learning algorithm called Bagging (Breiman 1996). Given a standard training set D of size n, Bagging generates m new training sets D_i, each of size n', by sampling from D uniformly and with replacement. Our approach is a bit different from Bagging, since we do not need to re-sample the training data.

Table 2 shows an example of the classifiers voting process to predict the label of a new comment. The columns correspond to the set of sub-classifiers and the label of a new comment predicted by each sub-classifier. The last row is the final output (i.e., predicted label of a new comment) of the composite classifier. In this example, we have 7 sub-classifiers

Table 2 An example of classifiers voting

Classifiers	Predicted Label
Sub-classifier 1	SATD comment
Sub-classifier 2	Without SATD
Sub-classifier 3	SATD comment
Sub-classifier 4	SATD comment
Sub-classifier 5	SATD comment
Sub-classifier 6	Without SATD
Sub-classifier 7	Without SATD
Composite classifier	SATD comment

trained from 7 different source project. 3 sub-classifiers predict the new comment as without SATD, while 4 sub-classifiers predict the new comment as SATD comment. So the final output of the composite classifier is “SATD comment” (i.e., the label predicted most often by its sub-classifiers).

4 Experiment Setup

In this section, we describe the experiment setup that we follow to evaluate the performance of our approach. The experimental environment is a computer equipped with Intel(R) Core(TM) i5-2410M CPU and 4GB RAM, running Windows 7 (64-bit). We first present our data collection and then present our evaluation metrics. The experiment results and research questions are presented in the next section.

4.1 Extracting Project Data

To conduct our study, we obtained the data provided by the authors in Maldonado and Shihab (2015), in addition to four additional projects provided by the same authors. Our data set contained comments from eight open source projects, namely ArgoUML, Columba, Hibernate, JEdit, JFreeChart, JMeter, JRuby, and SQuirrel SQL. These provided projects belong to different application domains, vary in size and the number of contributors, and vary in the number of comments. The provided dataset contained each comment as well as the classification of the comment, i.e., SATD comment or not.

In their prior work, the authors describe a number of heuristics that they applied to eliminate comments that are obviously impossible to be SATD comment (Maldonado and Shihab 2015). We refer the reader to the authors’ earlier work for details on the data, however, to make our paper self-contained, we also summarize the key aspects used to arrive at the dataset we used. The authors in Maldonado and Shihab (2015) apply the following heuristic rules to remove irrelevant comments:

1. Automatically generated comments with fixed format (i.e., Auto-generated constructor stubs, auto-generated method stubs and auto-generated catch blocks), which are inserted as part of code snippets by the IDE to generate constructors, methods, and try catch blocks, are removed.
2. Commented source code fragments do not contain SATD, thus they are removed.

Table 3 Summary of projects in our dataset

Project	Domain	Contributors	LoC	Comment Ratio
ArgoUML	UML Modeling Tool	87	926K	High
Columba	Email Client	10	155K	High
Hibernate	ORM Framework	314	703K	Low
JEdit	Text Editor	57	310K	Average
JFreeChart	Char Library	19	317K	High
JMeter	Performance Tester	41	354K	Average
JRuby	Ruby Interpreter	374	841K	Low
SQuirrel	SQL Client	40	708K	Average

3. Multiple single line comments that are related to each other are grouped into a block comment. This is done since the comments need to be analyzed to indicate whether they contain SATD or not, and having a part of a comment makes it difficult to analyze and gain any context.
4. Javadoc and licence comments rarely mention SATD, thus they were removed unless they contained at least one task annotation (i.e., “TODO:”, “FIXME:”, or “XXX:”) (Storey et al. 2008).

Leveraging the aforementioned heuristics and removing duplicate comments (i.e., if the text contents of multiple comments are completely the same, we only keep one comment) reduced the number of comments significantly. For example, in the ArgoUML project, the number of comments reduces from 67,716 to 5,606.

4.2 Manual Classification

After obtaining the filtered comments the authors in Maldonado and Shihab (2015) manually examined each comment and classified the comments based on whether the comment is a SATD comment or not.[3] To mitigate personal bias, the authors took a stratified sample of the full dataset, which is a sample that achieves a confidence level of 99% and a confidence interval of 5%. Then they got another independent person to classify the stratified sample of the comments and measured the level of agreement between the two manual classifiers. They report a high level of agreement (Cohen’s Kappa coefficient (Cohen 1968) of +0.81), which makes us confident in the classification of the provided dataset.

4.3 Data Summary

Table 3 presents the summary of 8 projects in our dataset. The columns correspond to the name of project, the domain of application, the number of contributors, the number of lines of code (LOC), and the comment quality. All the information in Table 3 are collected from Open Hub,[4] an on-line community and public directory that offers analytics, search services and tools for open source software. Note that “Comment Ratio” represents the ratio of

[3] The authors also classify the type of technical debt, but we do not leverage the type in our paper

[4] https://www.openhub.net

Table 4 Statistics of collected comments

Project	Release	Comments	after filtering	SATD	% of SATD	features
ArgoUML	0.34	67,716	5,606	1,149	20.5%	3,777
Columba	1.4	33,895	4,123	161	3.9%	2,687
Hibernate	3.3.2	11,630	2,543	428	16.8%	2,479
JEdit	4.2	16,991	4,683	234	5.0%	3,926
JFreeChart	1.0.19	23,474	2,502	109	4.4%	2,017
JMeter	2.10	20,084	4,184	318	7.6%	2,777
JRuby	1.4.0	11,149	3,731	462	12.4%	2,961
SQuirrel	3.0.3	27,474	4,498	226	5.0%	3,365

comment lines to code lines, where “High” means the project is well-commented, “Average” means the project contains the same ratio of comments as the majority of Java projects in Open Hub, and “Low” means the project contains low ratio of comments. We choose these projects because they apply in different domains, vary in size and the number of contributors, and also vary in the number of comments. Thus, we believe it offers a good opportunity to test whether our approach is generalizable.

Table 4 presents the statistics of the collected comments in our dataset. The columns correspond to the name of projects, the release version, the number of original comments, the number of comments after filtering with the heuristic rules, the number of SATD comments, the percentage of SATD comments, and the number of unique features in each project. It is important to note that the percentage of SATD comments in each project is quite low (i.e., the percentage ranges between 3.9% and 20.5%, with an average of 9.45%). We refer to this as the class imbalance phenomenon (He and Garcia 2009). Moreover, a recent survey by Vassallo et al. showed that software practitioners often use SATD to highlight poor implantation areas (Vassallo et al. 2016). This prior work shows that SATD is a practically important issue that has negative impact.

Finally, to enable others to use our techniques, we have published our source code and dataset on GitHub.[5]

4.4 Evaluation Metrics

Based on the training dataset, our approach will classify each comment in the testing dataset. We record four basic statistics based on the four possible classification results: *TP* (true positive) represents the number of comments that are classified as SATD comments when they truly are SATD comments; *FP* (false positive) represents the number of comments that are classified as SATD comments when they actually are without SATD. *FN* (false negative) represents the number of comments that are classified as without SATD when they actually are SATD comments; *TN* (true negative) represents the number of comments that are classified as without SATD when they truly are without SATD.

Using these four statistics, we compute Precision, Recall, and F1-score to evaluate the performance of our approach. They are defined as follows:

[5] https://github.com/tkdsheep/TechnicalDebt

Precision the proportion of comments that are correctly classified as SATD comments among those classified as SATD comments.

$$P = TP/(TP + FP) \tag{5}$$

Recall the proportion of SATD comments that are correctly classified among those true SATD comments.

$$R = TP/(TP + FN) \tag{6}$$

F1-score a summary measure that combines both precision and recall - it evaluates if an increase in precision (recall) outweighs a reduction in recall (precision).

$$F = (2 \times P \times R)/(P + R) \tag{7}$$

A higher precision indicates that the SATD comments identified by the approach are more likely to be true SATD comments, which can help developers detect SATD more effectively and avoid wasting time on manual checking comments without SATD. However, high precision does not guarantee that the approach can detect all SATD comments. On the other hand, a higher recall indicates that the approach can find more comments that truly contain SATD, which can help developers find more problems and avoid missing important technical debts. However, high recall does not guarantee that all SATD comments identified by the approach are true SATD comments.

In many cases, high precision indicates the sacrifice of recall, and vice versa (Han et al. 2006). Therefore, it depends on the practitioner's requirement and resources to decide the importance of precision and recall. For example, if developers want to detect SATD comments as much as possible and they have enough time or human resources to manually check and fix SATD comments identified by the automatic approach, then recall should be more important than precision. If developers only have limited time or human resources, and do not want to waste time on checking incorrectly identified SATD comments, then precision should be more important than recall.

Since there is a trade-off between precision and recall, it is difficult to compare the performance of several classification models by using only precision or recall alone (Han et al. 2006). For this reason, we compare the classification results using F1-score, which is a harmonic mean of precision and recall. This follows the setting used in prior software engineering papers (Arisholm et al. 2007; Rahman et al. 2012; Jiang et al. 2013; Shihab et al. 2013; Valdivia Garcia and Shihab 2014; Xia et al. 2016a, b; Xia et al. 2015a, b; Xu et al. 2016c). However, to make our paper self-contained, besides reporting F1-score, we also report the corresponding precision and recall in experiment results.

Due to space limitation, we report precision, recall and f1-score in separate tables. It is important to note that the precision, recall and f1-score are calculated from the same experiment results under the same experiment settings (e.g., in Table 5, our approach achieves precision of 0.804 on ArgoUML project, and the corresponding recall on ArgoUML project is 0.854, as shown in Table 6).

5 Experiment Results

In this section, we present our experiment results which answer a number of research questions. We present these questions and their answers in the following subsections.

Table 5 Precision of our approach and all baseline approaches

Target	Ours	Pattern	NBM	SVM	kNN	bestSub	NLP
ArgoUML	**0.804**	0.750	0.564	0.690	0.766	0.706	0.726
Columba	0.770	0.818	0.286	**0.857**	0.531	0.556	0.677
Hibernate	0.832	0.900	0.451	**1.000**	0.736	0.686	0.565
jEdit	0.702	**0.857**	0.309	0.000	0.475	0.422	0.473
JFreeChart	0.606	**0.833**	0.209	0.333	0.337	0.405	0.516
JMeter	0.797	0.778	0.328	**1.000**	0.620	0.522	0.503
JRuby	0.826	0.750	0.434	**0.900**	0.781	0.635	0.589
SQuirrel	0.708	0.471	0.250	**0.952**	0.538	0.436	0.325
Average.	0.756	**0.770**	0.354	0.717	0.598	0.537	0.547

The best precision is in bold

5.1 RQ1: How Effective is our Text Mining Approach in Identifying SATD? How Much Improvement can it Achieve Over Baseline Approaches?

Motivation Our goal is to provide an approach that can automatically classify SATD comments. However, for this approach to be useful, one of the first questions is to see how effective it is in performing its detection and whether it can perform as well as, or better than baseline approaches. Answering to this research question would shed light on how much our approach advances the state-of-the-art in the detection of SATD.

Approach To compare the performance of our approach, we choose 4 baseline approaches, which are listed below.

Baseline 1 (Pattern) The current state-of-the-art in detecting SATD is devised by Potdar and Shihab (2014). In their work, they collected data from 4 open source projects (i.e., Eclipse, Chromium OS, Apache HTTP Server, and ArgoUML) and manually read through 101,762 code comments to determine patterns that indicated SATD. In total, they identified 62 different comment patterns (i.e., keywords and phrases) that were noticed as recurrent in SATD comments. In their pattern based approach, a comment would be classified as SATD comment if and only if any one of the 62 patterns appears in the text content of the comment.

Table 6 Recall of our approach and all baseline approaches

Target	Ours	Pattern	NBM	SVM	kNN	bestSub	NLP
ArgoUML	0.854	0.028	0.712	0.030	0.330	0.834	**0.897**
Columba	**0.836**	0.070	0.820	0.047	0.203	0.820	0.688
Hibernate	0.748	0.072	**0.801**	0.056	0.318	0.769	0.610
jEdit	0.410	0.185	**0.631**	0.000	0.195	0.554	0.446
JFreeChart	**0.792**	0.050	0.703	0.020	0.307	0.762	0.485
JMeter	**0.766**	0.050	0.699	0.043	0.348	0.755	0.624
JRuby	**0.856**	0.055	0.76	0.023	0.232	0.841	0.580
SQuirrel	0.602	0.040	0.607	0.100	0.284	**0.672**	0.657
Average.	**0.733**	0.069	0.717	0.040	0.277	0.718	0.623

The recall is paired with the precision reported in Table 5 under the same conditions

Since they have published these patterns, we simply download them as our first baseline approach.

Baseline 2 (NBM, SVM and kNN) Since our approach is based on text mining, we also design another simple baseline approach using text mining techniques. In this baseline approach, each time we choose 1 project as a target project (i.e., for prediction), then we integrate all the comments in the other 7 source projects into one dataset that we train on. We directly build ONE classifier (i.e., without feature selection and classifiers voting strategy) on this training dataset to identify SATD comments in the target project. Note that we do not use feature selection in this baseline approach because in RQ3 we will discuss whether the voting strategy can improve performance, the performance of this baseline approach with feature selection will be discussed in RQ3. By default, we use Naive Bayes Multinomial (NBM) as the underlying classifier of our approach, while there are many other classification techniques. For example, Support Vector Machine (SVM) and k-Nearest Neighbor (kNN) are also widely used classification techniques (Han et al. 2006). So we also investigate their performance in our baseline approach. In our experiment, we use the implementation of NBM, SVM and kNN in Weka (Hall et al. 2009) with default settings, and we also implement our approach on top of Weka.

Baseline 3 (bestSub) In our approach, we composite sub-classifiers from different source projects to jointly predict whether a new comment in a target project is a SATD comment or not. Since the performance of different sub-classifiers may significantly vary, it is possible that a single sub-classifier trained on a single source project may achieve better performance than a composite classifier trained on various source projects. Thus, we investigate whether our approach outperforms the best individual sub-classifier - if it does not, then perhaps we are proposing a more complicated approach, when a much simpler one can be used, i.e., a single sub-classifier. Specifically, in this baseline approach, we train a sub-classifier with selected features for each source project. After that, we directly use this sub-classifier to predict the label of new comments in target project. We record the performance (i.e., F1-score, along with the corresponding precision and recall) of each sub-classifier, and pick out the best result achieved by one of the sub-classifiers to compare with the performance of our approach.

Baseline 4 (NLP) Most recently, Maldonado et al. (2017) proposed an approach to automatically identify the two most common types of SATD comments (i.e., design debt and requirement debt). In their work, they built a maximum entropy classifier based on natural language processing (NLP). Although they used the same dataset and same experiment setting with ours, we cannot directly compare their experiment results with ours, since they separately reported the precision, recall and F1-score of each type of SATD comment. Besides, our dataset also contains other types of SATD comment. To compare the performance of their approach with ours, we follow their work to do basic preprocessing (i.e., stemming and removing punctuation characters) and build the maximum entropy classifier to predict whether a comment contains SATD or not. Note that they didn't use feature selection or ensemble learning in their approach.

Finally, to check if the improvement of F1-score of our approach over the baseline approaches are statistically significant, we run the Wilcoxon signed-rank test (Wilcoxon 1945) at 95% significance level on two competing approaches. We consider that our approach statistically significantly improves a baseline approach at the confidence level of 95% if the Wilcoxon signed-rank test result (i.e., p-value) is less than 0.05. We also use the

Table 7 F1-score of our approach and all baseline approaches

Target	Ours	Pattern	NBM	SVM	kNN	bestSub	NLP
ArgoUML	**0.828**	0.054	0.629	0.057	0.461	0.756	0.802
Columba	**0.801**	0.129	0.424	0.090	0.294	0.649	0.682
Hibernate	**0.788**	0.133	0.577	0.106	0.444	0.693	0.587
jEdit	**0.518**	0.304	0.415	0.000	0.276	0.459	0.459
JFreeChart	**0.687**	0.093	0.323	0.037	0.321	0.524	0.500
JMeter	**0.781**	0.093	0.447	0.082	0.445	0.617	0.557
JRuby	**0.841**	0.102	0.552	0.046	0.358	0.724	0.584
SQuirrel	**0.651**	0.073	0.354	0.180	0.371	0.515	0.435
Average.	**0.737**	0.123	0.465	0.075	0.371	0.591	0.576
Improve.	–	499.19%	58.49%	882.67%	98.65%	24.70%	27.95%
p-value	–	4.7e-4	5.4e-4	7.8e-5	7.8e-5	0.019	0.010
Cliff's delta	–	1.00	0.91	1.00	1.00	0.63	0.67

The best F1-score is in bold

The F1-score is calculated using the precision and recall in Tables 5 and 6

Cliff's delta (δ) to quantify the amount of difference between two approaches. The amount of difference is considered negligiable ($|\ \delta\ | < 0.147$), small ($0.147 \leq |\ \delta\ | < 0.33$), medium ($0.33 \leq |\ \delta\ | < 0.474$), or large ($|\ \delta\ | \geq 0.474$).

Results Tables 5–7 present the experiment results (i.e., precision, recall and F1-score) of our approach compared with baseline 1 (Pattern), baseline 2 (NBM, SVM and kNN), baseline 3 (bestSub) and baseline 4 (NLP), respectively. For each target project, the best results of precision, recall and F1-score are highlighted in bold. The last 3 rows in Table 7 present the improvement of F1-score of our approach over other baseline approaches and the corresponding p-value.

As for baseline 3, since we only present the performance of the best sub-classifier, to make our paper self-contained, we also present the full results of baseline 3 in Tables 8, 9 and 10, including the precision, recall and F1-score of using a sub-classifier trained on a single source project to predict the label of a new comments in a target project. Due to

Table 8 Precision when using a single source project as training data

Target	Argo.	Col.	Hib.	JE.	JF.	JM.	JR.	SQ.	Our	Improv.
Argo.	–	0.657	0.647	0.659	0.699	0.706	0.642	0.692	0.804	13.88%
Col.	0.249	–	0.438	0.360	0.349	0.556	0.360	0.411	0.770	38.49%
Hib.	0.410	0.588	–	0.581	0.629	0.686	0.575	0.630	0.832	21.28%
JE.	0.252	0.313	0.327	–	0.379	0.348	0.422	0.416	0.702	66.35%
JF.	0.209	0.338	0.368	0.291	–	0.405	0.397	0.246	0.606	49.63%
JM.	0.297	0.436	0.461	0.522	0.512	–	0.477	0.482	0.797	44.38%
JR.	0.428	0.610	0.635	0.534	0.513	0.624	–	0.507	0.826	30.08%
SQ.	0.205	0.286	0.353	0.318	0.435	0.436	0.382	–	0.708	62.39%
Avg.	0.293	0.461	0.461	0.466	0.502	0.537	0.465	0.483	0.756	40.78%

Table 9 Recall when using a single source project as training data

Target	Argo.	Col.	Hib.	JE.	JF.	JM.	JR.	SQ.	Our	Improv.
Argo.	–	0.739	0.752	0.834	0.824	0.747	0.724	0.804	0.854	2.40%
Col.	0.820	–	0.805	0.820	0.695	0.781	0.797	0.797	0.836	1.95%
Hib.	0.658	0.700	–	0.769	0.714	0.700	0.706	0.724	0.748	−2.73%
JE.	0.554	0.415	0.446	–	0.410	0.328	0.503	0.456	0.410	−25.99%
JF.	0.752	0.752	0.762	0.564	–	0.743	0.762	0.554	0.792	3.94%
JM.	0.677	0.713	0.770	0.755	0.702	–	0.702	0.745	0.766	1.46%
JR.	0.742	0.809	0.841	0.611	0.679	0.836	–	0.548	0.856	1.78%
SQ.	0.632	0.577	0.652	0.672	0.632	0.562	0.582	–	0.602	−10.42%
Avg.	0.691	0.672	0.718	0.718	0.665	0.671	0.682	0.661	0.733	2.09%

The recall is paired with the precision reported in Table 8 under the same conditions

space limitation, the name of each source project is abbreviated, as shown in the first row of each table. The first column corresponds to each target project, and the last 2 columns present the result of our approach (i.e., ensemble all sub-classifiers on source projects) and the improvement over the best sub-classifier. For each target project, the results of the best sub-classifier are highlighted in bold. For example, when choosing Columba as the target project, the sub-classifier trained on JMeter achieves the best F1-score of 0.649.

The F1-score achieved by our approach ranges between 0.518 - 0.841, with an average of 0.737. In comparison, the average F1-score achieved by Potdar and Shihab's pattern based approach, NBM, SVM, kNN, bestSub and Maldonado et al.'s NLP-based approach are 0.123, 0.465, 0.075, 0.371, 0.591 and 0.576, respectively. Among all the classification techniques in baseline 2, NBM achieves the best performance in terms of F1-score, with an average of 0.465. That is why we choose NBM as the default underlying classifier in our approach. We notice that, in each target project, the F1-score achieved by our approach is higher than any other baseline approach. In summary, on average our approach improves the F1-scores over Potdar and Shihab's pattern based approach, NBM, SVM, kNN, bestSub and Maldonado et al.'s NLP-based approach by 499.19%, 58.49%, 882.67%, 98.65%, 24.70% and 27.95%, respectively. When comparing our approach with each baseline approach, the

Table 10 F1-score when using a single source project as training data

Target	Argo.	Col.	Hib.	JE.	JF.	JM.	JR.	SQ.	Our	Improv.
Argo.	–	0.695	0.696	0.736	0.756	0.726	0.681	0.744	0.828	9.52%
Col.	0.382	–	0.567	0.500	0.465	0.649	0.496	0.543	0.801	23.42%
Hib.	0.505	0.639	–	0.662	0.668	0.693	0.633	0.674	0.778	12.27%
JE.	0.347	0.357	0.377	–	0.394	0.338	0.459	0.435	0.518	12.85%
JF.	0.328	0.466	0.497	0.384	–	0.524	0.522	0.340	0.687	31.11%
JM.	0.413	0.541	0.576	0.617	0.592	–	0.568	0.585	0.781	26.58%
JR.	0.543	0.696	0.724	0.570	0.584	0.714	–	0.527	0.841	16.16%
SQ.	0.310	0.383	0.458	0.432	0.515	0.491	0.462	–	0.651	26.41%
Avg.	0.404	0.540	0.556	0.557	0.568	0.591	0.546	0.550	0.737	24.70%

The F1-score is calculated using the precision and recall in Tables 8 and 9 corresponding p-value (all less than 0.05) and cliff's delta (all larger than 0.474) also indicate that our approach significantly improves F1-score over all the other baseline approaches by a substantial margin.

It is also important to note that, Potdar and Shihab's pattern based approach achieves the best performance in terms of precision, with an average of 0.770. On average, our approach achieves precision of 0.756, which is very close to their result. However, the recall of their approach (i.e., 0.069) is much lower than that of our approach (i.e., 0.733). This is the case since the number of patterns in their approach is still not enough to cover all important features of SATD comments, thus identifying only a small subset of all the SATD comments in each target project. Similarly, SVM also achieves relatively high precision (i.e., 0.717) but with the lowest recall (0.040). One reason is that only a small proportion of comments are with SATD (e.g., 3.9% comments in Columba project are SATD comments), which makes SVM prefer to non-SATD comments, thus misclassifying most SATD comments as non-SATD comments.

Finally, since Potdar and Shihab's pattern based approach can achieve a high precision, we would like to investigate whether it is worthwhile to combine these patterns with our approach. By manually checking the classification results of the pattern based approach and our approach, we find that most of the SATD comments identified by the pattern based approach can also be identified by our approach. Specifically, for each target project, the number of SATD comments that can only be identified by the pattern based approach while cannot be identified by ours is less than 10. Thus, we believe that there is no need to combine these patterns with our approach.

For each target project, our approach achieves the best performance in terms of F1-score. On average across the 8 target projects, our approach achieves F1-score of 0.737, which significantly improves all the other baseline approaches in a substantial margin.

5.2 RQ2: Does Feature Selection Improve the Performance of our Approach?

Motivation In our approach, for each source project, we apply feature selection to preprocess comments, and then use the selected features to train a classifier. By default, we only keep the top 10% features with the highest feature selection scores. But the performance of our approach may vary if different percentage of features are selected. Besides, since a large

Table 11 Precision of our approach with varying percentages of selected features

Target	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Argo.	0.801	**0.804**	0.770	0.749	0.717	0.707	0.704	0.692	0.697	0.691	0.694
Col.	**0.797**	0.770	0.731	0.611	0.573	0.575	0.593	0.571	0.563	0.528	0.520
Hib.	**0.878**	0.831	0.787	0.768	0.724	0.699	0.691	0.665	0.662	0.651	0.663
JE.	**0.760**	0.702	0.564	0.528	0.533	0.522	0.515	0.475	0.482	0.440	0.416
JF.	**0.608**	0.606	0.535	0.487	0.462	0.402	0.411	0.428	0.421	0.426	0.439
JM.	**0.802**	0.797	0.718	0.698	0.594	0.575	0.575	0.553	0.541	0.537	0.527
JR.	**0.864**	0.826	0.737	0.722	0.693	0.681	0.667	0.669	0.672	0.675	0.674
SQ.	**0.734**	0.708	0.635	0.563	0.545	0.537	0.534	0.532	0.502	0.489	0.477
Avg.	**0.781**	0.756	0.685	0.641	0.605	0.587	0.586	0.573	0.568	0.555	0.551

The best precision is in bold

Table 12 Recall of our approach with varying percentages of selected features. The recall is paired with the precision reported in Table 11 under the same conditions

Target	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Argo.	**0.883**	0.854	0.812	0.773	0.780	0.748	0.749	0.728	0.699	0.693	0.684
Col.	0.797	**0.836**	0.828	0.797	0.797	0.805	0.820	0.812	0.805	0.797	0.797
Hib.	0.745	**0.748**	0.745	0.745	0.743	0.703	0.711	0.716	0.711	0.716	0.716
JE.	0.374	0.410	0.431	0.431	0.456	0.492	**0.518**	0.492	0.492	0.508	0.472
JF.	0.752	**0.792**	0.762	0.762	0.723	0.713	0.733	0.703	0.713	0.713	0.713
JM.	**0.777**	0.766	0.741	0.762	0.748	0.723	0.720	0.720	0.709	0.713	0.695
JR.	0.713	**0.856**	0.828	0.833	0.849	0.836	0.830	0.812	0.825	0.815	0.815
SQ.	0.617	0.602	0.597	0.602	**0.657**	0.612	0.587	0.582	0.557	0.567	0.567
Avg.	0.707	**0.733**	0.718	0.713	0.719	0.704	0.709	0.696	0.689	0.690	0.682

The best precision is in bold

number of features are removed during the feature selection phase, it is possible that some features with important semantic information are also removed, which may impact the performance of our approach. Therefore, we investigate whether feature selection improves the performance of our approach.

Approach To answer this research question, we vary the number of selected features from 10% to 100% (with a step of 10%) of the total number of features and compute the corresponding precision, recall and F1-score of our approach. Additionally, we also investigate the performance of our approach when selecting 5% features. Notice that feature selection is actually useless when we select 100% features. To further investigate the impact of feature selection, we also compare the performance of our approach with and without feature selection (i.e., selecting 10% features vs. 100% features).

Results Tables 11, 12 and 13 present the precision, recall and F1-score of our approach on each target project when selecting different percentage of features. The results show that, for each target project, our approach achieves the best performance in terms of F1-score when selecting the top 5% or 10% features. We also plot the resultant F1-scores for

Table 13 F1-score of our approach with varting percentages of selected features

Target	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Argo.	**0.840**	0.828	0.791	0.761	0.747	0.727	0.726	0.709	0.698	0.692	0.689
Col.	0.797	**0.801**	0.777	0.692	0.667	0.671	0.689	0.671	0.662	0.636	0.630
Hib.	**0.806**	0.788	0.766	0.756	0.733	0.701	0.701	0.690	0.685	0.682	0.689
JE.	0.502	**0.518**	0.488	0.475	0.492	0.507	0.517	0.484	0.487	0.471	0.442
JF.	0.673	**0.687**	0.629	0.595	0.564	0.514	0.527	0.532	0.529	0.533	0.543
JM.	**0.789**	0.781	0.729	0.729	0.662	0.641	0.639	0.626	0.613	0.613	0.599
JR.	0.781	**0.841**	0.780	0.773	0.763	0.750	0.740	0.733	0.741	0.738	0.738
SQ.	**0.670**	0.651	0.615	0.582	0.596	0.572	0.559	0.556	0.528	0.525	0.518
Avg.	0.732	**0.737**	0.697	0.670	0.653	0.635	0.637	0.625	0.618	0.611	0.606

The best F1-score is in bold

The F1-score is calculated using the precision and recall in Tables 11 and 12

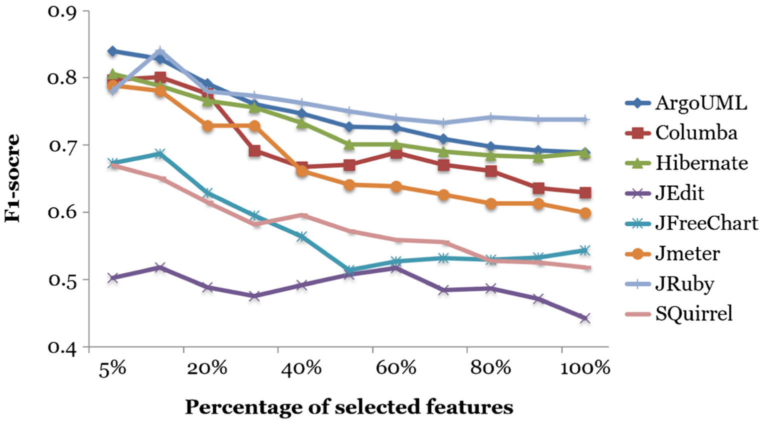


Fig. 2 F1-score of each target project with different percentage of selected features

each target project in Fig. 2. From Fig. 2, we can notice a general trend: on average across the 8 target projects, the F1-score decreases when we increase the percentage of selected features. One evident exception is JEdit project. Unlike other projects, the F1-score curve for JEdit fluctuates. Especially when selecting 60% features, our approach achieves F1-score of 0.517 on JEdit, which is very close to the result when selecting 10% features (i.e., 0.518). One reason is that the feature distribution in JEdit is quite different from other projects. Thus, some features with important semantic information for JEdit are mistakenly removed during feature selection, since they do not contribute much to distinguish SATD comments in training data.

Table 14 presents the comparison of precision, recall and F1-score of our approach with and without feature selection. When applying our approach without feature selection, the F1-score ranges between 0.442 and 0.738, with an average of 0.606. In comparison, the full version of our approach improves F1-score over the version without feature selection in every target project. On average, our approach improves F1-score by 21.62%. Similarly, on average, our approach improves precision and recall by 37.21% and 7.48%, respectively.

Table 14 Comparison of precision, recall and F1-score of our approach with and without feature selection

Target	Precision			Recall			F1-score		
	NoFS	Our	Improv.	NoFS	Our	Improv.	NoFS	Our	Improv.
Argo.	0.694	0.804	**15.85%**	0.684	0.854	**24.85%**	0.689	0.828	**20.17%**
Col.	0.520	0.770	**48.08%**	0.797	0.836	**4.89%**	0.630	0.801	**27.14%**
Hib.	0.663	0.832	**24.49%**	0.716	0.748	**4.47%**	0.689	0.788	**14.37%**
JE.	0.416	0.702	**68.75%**	0.472	0.410	**−13.14%**	0.442	0.518	**17.19%**
JF.	0.439	0.606	**38.04%**	0.713	0.792	**11.08%**	0.543	0.687	**26.52%**
JM.	0.527	0.797	**51.23%**	0.695	0.766	**10.22%**	0.599	0.781	**30.38%**
JR.	0.674	0.826	**22.55%**	0.815	0.856	**5.03%**	0.738	0.841	**13.96%**
SQ.	0.478	0.708	**48.12%**	0.567	0.602	**6.17%**	0.518	0.651	**25.68%**
Avg.	0.551	0.756	**37.21%**	0.682	0.733	**7.48%**	0.606	0.737	**21.62%**

The improvement is in bold

We also apply Wilcoxon signed-rank test to check if the improvement in the F1-score of our approach over the approach without feature selection is statistically significant. We find that the corresponding p-value is 0.023 (less than 0.05), which indicates that feature selection can significantly improve the F1-score of our approach.

We also notice that, when JEdit is chosen as the target project, using feature selection cannot improve recall. The recall of our approach decreases by 13.14% compared to the version without feature selection. To gain insight into this case, we manually checked the SATD comments that are not successfully identified by our approach. We find that some important features (e.g., "nasty", "wtf" and "fudge"), which are highly related to SATD comments in JEdit project, are actually removed during the feature selection phase. However, if not using feature selection, it achieves a low precision of 0.416 in JEdit project. This is because that a large number of noisy features are introduced, which leads to much more comments being incorrectly identified as SATD comments. Although applying feature selection may decrease recall, the improvement on precision is much more significant (i.e., from 0.416 to 0.702 in JEdit project, with an improvement of 68.75%). Thus, considering the trade-off between precision and recall, feature selection is still useful in general.

For each target project, our approach achieves the best performance in terms of F1-score when selecting the top 5% or 10% features. In general, our approach achieves better performance when selecting less features. On average, our approach improves F1-score over the version without feature selection by 21.62%. This means that our approach can provide superior performance, while reducing the amount of data we need to train on.

5.3 RQ3: Can Classifiers Voting Improve the Performance of our Approach?

Motivation In our approach, we train an individual sub-classifier for each source project, and then combine them as a composite classifier to vote for the comments in the target project. However, when using just one source project as training data, the corresponding sub-classifier may bias to some domain-specific features which is negligible in the target project. Another way to leverage training data is integrating all comments from all the 7 source projects (we leave 1 target project as the test set) into a whole dataset to train a single classifier, in which voting strategy is not needed. Therefore, we investigate whether the voting strategy improves the performance of our approach.

Approach To answer this research question, we compare the performance of our approach with and without voting strategy. More specifically, we integrate all the comments in every source project into a whole dataset for training. Different from the baseline approach in RQ1, here we apply feature selection (with the same setting of our approach) before training classifier. We use NBM as the underlying classifier.

Results Table 15 presents the comparison of precision, recall and F1-score of our approach with and without classifier voting. When applying our approach without voting strategy, the F1-score ranges between 0.447 and 0.732, with an average of 0.590. In comparison, for each target project, the full version of our approach improves the F1-score over the version without classifiers voting. On average, our approach improves F1-score by 24.92%. Similar to the situation in RQ2 and RQ3, our approach sacrifice a bit of recall to achieve a significant improvement in precision. We also apply Wilcoxon signed-rank test to check if the improvement in the F1-score of our approach over the approach without voting strategy

Table 15 Comparison of precision, recall and F1-score of our approach with and without voting strategy

Target	Precision			Recall			F1-score		
	No-V.	Our	Improv.	No-V.	Our	Improv.	No-V.	Our	Improv.
Argo.	0.701	0.804	**14.69%**	0.766	0.854	**11.49%**	0.732	0.828	**13.11%**
Col.	0.466	0.770	**65.24%**	0.844	0.836	**−0.95%**	0.600	0.801	**33.5%**
Hib.	0.571	0.832	**45.71%**	0.767	0.748	**−2.48%.**	0.655	0.788	**20.31%**
JE.	0.423	0.702	**65.96%**	0.595	0.410	**−31.09%**	0.495	0.518	**4.65%**
JF.	0.379	0.606	**59.89%**	0.762	0.792	**3.94%**	0.507	0.687	**35.50%**
JM.	0.496	0.797	**60.69%**	0.727	0.766	**5.36%**	0.590	0.781	**32.37%**
JR.	0.602	0.826	**37.21%**	0.815	0.856	**5.03%**	0.693	0.841	**21.36%**
SQ.	0.350	0.708	**102.29%**	0.617	0.602	**−2.43%**	0.447	0.651	**45.64%**
Avg.	0.499	0.756	**51.50%**	0.737	0.733	**−0.54%**	0.590	0.737	**24.92%**

The improvement is in bold

is statistically significant. We find that the corresponding p-value is 0.010 (less than 0.05), which indicates that voting strategy can significantly improve the F1-score of our approach.

By manually checking the classification results of each sub-classifier, we find that different sub-classifiers have different "preferences". For example, there is an SATD comment talking about "a hack", and the sub-classifiers trained from the ArgoUML project cannot identify this SATD comment while the other sub-classifiers successfully identify it. One reason is that the training data in the ArgoUML project does not contain necessary information to let the sub-classifier learn that "hack" has a strong possibility of indicating SATD comments. However, by combining these sub-classifiers, which are complementary to each other, they no longer bias to certain kind of SATD comments, thus improving the performance.

For each target project, our approach improves F1-score over the version without classifiers voting. On average, our approach improves F1-score by 24.92%

5.4 RQ4: What's the top-k Features After Feature Selection?

Motivation In our approach, we use feature selection to select features that contain more information for classification. So it gives us an opportunity to shed light on what words lead to a comment being classified as SATD. Answering this research question can help us better understand the nature of SATD, and provide an intuition behind how our approach works to identify SATD comments.

Approach To answer this research question, for each project, we apply feature selection and present the top 30 features with the highest information gain scores.

Results Table 16 presents the top 30 features (i.e., words) after feature selection on each project. Note that the information gain score indicates the amount of information carried by a word for predicting a label (i.e., with or without SATD). Therefore, a word with high information gain score does not necessarily mean that it indicates SATD. Thus, we manually

Table 16 Top 30 features after feature selection on each project

ArgoUML	Columba	Hibernate	JEdit	JFreeChart	JMeter	JRuby	SQuirrel
todo	**todo**	**todo**	thi	**todo**	**todo**	**todo**	**todo**
thi	author	thi	should	**fixm**	thi	**fixm**	thi
should	fdietz	should	**hack**	thi	should	thi	some
need	**fixm**	better	note	**implement**	**hack**	sss	wai
tfm	**implement**	here	**workaround**	properli	perhap	line	**hack**
param	thi	wai	need	state	why	should	should
return	hubm	**yuck**	method	shift	realli	here	not
see	should	**ugli**	here	**ugli**	wai	probabl	that
us	better	would	could	timezon	correct	**hack**	**workaround**
that	**workaround**	realli	work	**hack**	appear	but	bit
author	work	**hack**	that	realli	best	need	**implement**
here	want	**workaround**	but	idea	make	not	us
modelext	param	**fix**	and	could	improv	better	problem
creat	dont	move	elimin	method	more	mai	and
why	wai	around	us	interfac	doe	into	dialect
implement	**hack**	need	probabl	block	what	make	with
model	that	doe	namespace	attribut	**fix**	somewher	what
and	us	perhap	**todo**	render	notus	gross	more
can	nonjavadoc	**fixm**	**stupid**	should	class	from	author
code	manual	onc	wai	make	could	move	like
what	webstart	work	class	half	seem	some	out
uml	real	packag	**xxx**	ye	need	effici	why
method	though	first	**implement**	probabl	**fixm**	wai	jason
move	there	into	resolv	strictly	consid	check	can
where	becaus	bug	**fix**	think	effici	right	need
better	here	bit	rewritten	wherea	mayb	would	method
not	extend	depend	then	**ill**	allow	realli	db
more	know	why	code	need	error	could	when
but	dialog	done	**broken**	here	better	go	code
instanc	replac	**ineffici**	realli	swt	here	dont	handl

Frequent features are in bold

check each word in Table 16 and pick out the words (highlighted in bold) that appears frequently in SATD comments.

From the results, we first find that different projects may share some common patterns (i.e., keywords) that indicate SATD comments, such as “todo”, “fixme”, “workaround”, “implement”, “hack”, etc. But the frequency of these words in different projects may vary. For example, for a temporary fix, some developers prefer to use the word “hack”, while some other developers prefer to use the word “workaround”. Besides, it is important to note that these words may also appear in non-SATD comments. For example, when the word “implement” appears in non-SATD comments, it may indicate that the developer has written the code to implement some functions (e.g., “Implements backspace functionality”). While in SATD comments, the word “implement” usually indicates that the developer needs to implement some functions but hasn't done it yet(e.g., "Bunch of methods still not implemented").

We also find that when writing SATD comments, some developers prefer to use sentimental words (e.g., yuck, ugly, stupid, ill, etc). By manually reading these SATD comments, we find that in most cases, developers are forced to make quick fixes or implementations (e.g., as indicated by the comment "This is really ugly, but necessary"). Also, these words rarely appear in non-SATD comments unless developers want to state that they want to avoid writing bad quality code (e.g., "guard against something really stupid").

We find some common patterns that appear in SATD comments of different projects. We also find that when writing SATD comments, some developers prefer to express their opinions in an emotional way.

5.5 RQ5: What's the Performance of our Approach if Using Comments Within the Target Project as Training Data?

Motivation Our approach simulates a realistic scenario that the classification system is trained on existing comments from old source projects and then a new target project appears. If we deploy such a system on a new target project, more and more comments would be identified by the system over time, and new SATD comments would also emerge. In this situation, we would like to know whether it is worth retraining the system to account for all these new comments, or to just stick with the default training set. Answering this research question would provide useful guidance for practitioners on how to leverage new data to improve the classification system.

Approach To answer this question, for each target project, we perform stratified 10-fold cross-validation. Specifically, given a target project, we first randomly shuffle the dataset and equally divide the dataset into 10 folds, where each fold contains nearly equal proportion of comments belonging to each label (i.e., with or without SATD). Then we perform 10 evaluation rounds with different testing dataset; in each round, 9 folds are used as training dataset, and the remaining one fold is used as testing dataset. We aggregate the results of the 10 evaluation rounds and report the overall performance. Stratified cross-validation is a standard evaluation setting, which is widely used in software engineering studies (Valdivia Garcia and Shihab 2014; Shihab et al. 2013; Xia et al. 2013). Since stratified 10-fold cross-validation involves randomness, to increase the confidence of the results, we repeat it 10 times and report the average results.

Results Table 17 presents the results of precision, recall and F1-score when using comments within the target project as training data. Specifically, "Within" means we only use comments within the target project to train a classifier (using NBM and feature selection). "Our" means our original approach, which only uses comments from the other 7 source projects as training data to train 7 sub-classifiers (using NBM and feature selection) and vote to decide the label of a comment in testing dataset. "W+O" means we combine the classifier trained on the target project with the other 7 sub-classifiers trained on 7 source projects so that there would be 8 sub-classifiers when voting. For each target project, the highest precision, recall and F1-score are highlighted in bold.

The results show that the "Within" approach achieves the highest average recall (i.e., 0.770). One reason is that the classifier trained on the target project can learn more specific SATD patterns that rarely appear in other projects. However, the number of comments in only 1 project is still quite small, which is difficult to train a robust classifier. So the average precision (i.e., 0.639) of the "Within" approach is still quite lower than the other 2 approaches. On the other hand, when combining "Within" approach with our approach (i.e., "W+O"), we can achieve the highest precision, with a bit sacrifice of recall. And the F1-score of "W+O" approach even outperforms our original approach. This indicates that it is worthwhile to retrain the system to account for all these new comments.

We find that using comments in the target project to train a new sub-classifier and combine it with our original approach can further improve F1-score.

5.6 RQ6: How Much Memory and Time Does it Take for our Approach to run?

Motivation Although our approach achieves better results in terms of F1-score when compared with other baseline approaches, we haven't investigated whether our approach can run efficiently in a reasonable cost of memory and time. Thus, we investigate how much memory and time it costs our approach to run.

Approach For memory cost, we mainly discuss how many words are reduced after applying feature selection. We record and compare the total number of words (i.e., duplicate words are also counted in) in each project before and after feature selection. For time cost, we record the model building and prediction time of our approach and the baseline approaches. Model building time refers to the time it takes to convert the training data into the composite classifier. Prediction time refers to the time it takes to predict the label of a source code comment.

Result Table 18 presents the total number of words in each project before and after feature selection. The reduction of the total number of words ranges between 32.57% - 76.85%, with an average of 55.17%. Since than half of the words are reduced after feature selection, we believe our approach can efficiently reduce memory cost.

Tables 19 and 20 presents the model building and prediction time for each of the 8 target projects. We notice that Potdar and Shihab's pattern-based approach has the fastest model building time (i.e., 0.001s on average), since it only needs reading 62 patterns from file to memory. The model building time of our approach is relatively slow (i.e.,

Table 17 Results of precision, recall and F1-score when using comments within the target project as training data

Metrics	App.	Argo.	Col.	Hib.	JE.	JF.	JM.	JR.	SQ.	Avg.
Precision	Within	0.653	0.618	0.726	0.617	0.470	0.728	0.749	0.548	0.639
	Our	0.804	0.770	0.832	0.702	0.606	0.797	0.826	0.708	0.756
	W+O	**0.813**	**0.827**	**0.883**	**0.789**	**0.670**	0.871	0.867	**0.777**	**0.812**
Recall	Within	0.766	**0.898**	**0.780**	**0.621**	0.762	**0.777**	0.817	**0.736**	**0.770**
	Our	**0.854**	0.836	0.748	0.410	**0.792**	0.766	**0.856**	0.602	0.733
	W+O	0.819	0.820	0.719	0.385	0.743	0.745	0.817	0.572	0.702
F1-score	Within	0.705	0.732	0.752	**0.619**	0.581	0.751	0.782	0.628	0.693
	Our	**0.828**	0.801	0.788	0.518	0.687	0.781	**0.841**	0.651	0.737
	W+O	0.816	**0.824**	**0.792**	0.517	**0.704**	**0.803**	**0.841**	**0.659**	**0.745**

The best precision, recall, and F1 scores are in bold

Table 18 Total number of words in each project before and after feature selection

FS	Argo.	Col.	Hib.	JE.	JF.	JM.	JR.	SQ.	Avg.
Before	73,179	31,258	28,823	29,665	20,143	40,601	31,977	57,541	39,148
After	49,347	11,055	73,83	14,709	4,663	10,918	13,621	28,699	17,549
Reduce	32.57%	64.63%	74.39%	50.42%	76.85%	73.11%	57.40%	50.12%	55.17%

22.884s on average), since feature selection is time-consuming. However, we believe it is still acceptable since our model does not need to be updated all the time, and it can be applied to label many comments in new projects. On the other hand, our approach has the fastest prediction time. More specifically, the prediction time of text mining-based baseline approaches are much longer than our approach. One reason is that after feature selection, our approach can train classifiers using a small subset of features, making it more lightweight in prediction.

Our approach can reduce total number of words by 55% on average. The model building and prediction time for our approach are also reasonable.

6 Discussion

In this section, we first discuss the implications of our study. Then we discuss threats to validity.

6.1 Implications for Practitioners

For project managers, our approach can help them evaluate project quality more comprehensively, thus making better decisions (e.g., whether to release or not). For developers, our approach can remind them the existence of historical SATD in case of being left forgotten. For newcomers who are just taking over previous work from other developers, our approach can also inform them of existing SATD comments so that they can handle potential issues more efficiently.

Our approach needs historical comments for training and practitioners can easily collect comments from software repositories. But in practice, it is possible that a project does not have enough comments with correct label (i.e., with or without SATD). To solve this problem, our experiment simulates the realistic scenario that the system is trained on existing comments from old projects and then a new project appears for testing. The experiment results show that SATD comments in different projects may share some common patterns

Table 19 Model building time for our approach, Potdar and Shihab's pattern-based approach, NBM baseline, SVM baseline and kNN baseline (in seconds)

App.	Argo.	Col.	Hib.	JE.	JF.	JM.	JR.	SQ.	Avg.
Our	24.466	24.307	23.417	21.268	23.750	22.561	21.970	21.333	22.884
Pattern	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
NBM	3.979	2.078	1.561	1.265	1.070	1.014	1.061	0.999	1.628
SVM	47.754	49.582	48.580	49.467	54.933	47.247	50.059	48.194	49.477
kNN	4.012	1.987	1.108	1.061	1.076	1.326	1.031	0.967	1.571

Table 20 Model prediction time for our approach, Potdar and Shihab's pattern-based approach, NBM baseline, SVM baseline and kNN baseline (in seconds)

App.	Argo.	Col.	Hib.	JE.	JF.	JM.	JR.	SQ.	Avg.
Our	0.031	0.001	0.015	0.015	0.016	0.016	0.016	0.001	0.014
Pattern	0.064	0.047	0.094	0.001	0.001	0.016	0.015	0.016	0.032
NBM	0.142	0.078	0.036	0.048	0.015	0.016	0.016	0.016	0.046
SVM	5.727	5.243	3.201	6.053	3.121	5.414	4.383	6.120	4.908
kNN	35.195	28.098	25.589	28.539	18.134	32.161	23.543	27.880	27.392

and our approach is practical in such scenario. Also, our approach is scalable in use. For example, with more and more SATD comments identified by the system, practitioners can choose to retrain the model by adding these new SATD comments.

6.2 Implications for Researchers

Our approach pushes the frontier of research in identification of SATD. While most of the current empirical work (e.g., Wehaibi et al. (2016) and Bavota and Russo (2016)) of SATD are based on the 62 patterns summarized in Potdar and Shihab's work (Potdar and Shihab 2014), our experiment results show that Potdar and Shihab's pattern based approach only achieves an average recall of 0.069, which indicates that a large number of SATD comments cannot be identified by these patterns. Compared with that, our approach can identify more SATD comments while preserving a high precision. Thus, we believe that our work highlights an opportunity for further study to gain more insights into SATD.

6.3 Threats to Validity

Threats to internal validity relates to errors in our code and personal bias in manual classification of training comments. To reduce errors in our code, we have double checked and fully tested our code, still there could be errors that we did not notice. To reduce personal bias in manual classification of code comments, the authors in Maldonado and Shihab (2015) took a statistically significant sample of classified comments and asked an independent masters student, who is not an author of their paper, to manually classify them. Then, reported a high level of agreement between the classification given by the two different students, i.e., Cohen's kappa coefficient of +0.81. This gives us high confidence in the dataset provided to us and used in our paper.

Threats to external validity relates to the quantity and quality of our dataset. To guarantee quantity and quality of our dataset, we use 8 open source projects that vary in the number of comments. In total, we have analyzed 31,870 comments. However, all the 8 projects are developed by open source communities, it is still unclear whether our approach is generalizable when applying to projects in a company. Since open source communities are highly transparent and developers are usually forced to do a lot of communication through the source code (as they are distributed), they are more likely to admit technical debt in comments. In contrast, it is possible that developers in a company are hesitant to admit technical debt in the source code, as it might put them in bad light - thus affecting both salaries and job security in general. So in this paper, we only focus on the scope of "Open Source Projects". Besides, although 8 projects are analyzed, all of them are medium-sized desktop applications implemented in Java. There are still many other projects in different domains not considered in our paper. In the future, we plan to reduce this threat further by analyzing even more comments from additional software projects.

Threats to construct validity refers to the suitability of our evaluation metrics. We use precision, recall and F1-score which are also used by past studies to evaluate the performance of various automated software engineering techniques (Arisholm et al. 2007; Rahman et al. 2012; Jiang et al. 2013; Shihab et al. 2013; Valdivia Garcia and Shihab 2014). Thus, we believe there is little threat to construct validity.

7 Related Work

In our work, we apply text mining to detect SATD from source code comments. Therefore, we divide the related work into three categories: technical debt, source code comments and text mining in software engineering.

7.1 Technical Debt

There have been a number of studies on detection and management of technical debt. For example, Zazworka et al. evaluated how the technical debt list can be populated by developers through a common template, and how existing tool approaches can help to identify certain types of debt. The technical debt items identified by both approaches were categorized under defect debt, design debt, documentation debt, testing debt and usability debt. They found that code smells, automatic static analysis and code metrics are effective to automatically identify defect debt and a partial set of files with design debt (Zazworka et al. 2013). In their follow-on work, they investigated how design debt, in the form of god classes, affects the maintainability and correctness of software products. Their findings suggest that technical debt has a negative impact on software quality (Zazworka et al. 2011). Kruchten et al. presented an in-depth understanding of the definition of technical debt, the limits of the metaphor, and the criteria to discriminate what is technical debt and not (Kruchten et al. 2013). For example, they argue that technical debt is not simply bad quality and it can be a wise investment. Guo et al. explored the effect of technical debt by tracking a single delayed maintenance task in a real software project throughout its lifecycle and simulated how explicit technical debt management might have changed project outcomes (Guo et al. 2011). They argued that software manager should perform detailed cost-benefit analysis of the technical debt item when they make their decisions. Seaman et al. (2012) and Brown et al. (2010) summarized a number of properties of technical debt, including visibility, value, origin of debt, impact of debt, etc. They also investigated how can technical debt assist in project decision-making.

Recently, Potdar and Shihab (2014) proposed the concept of self-admitted technical debt (SATD), which considers debt that is intentionally introduced (e.g., quick or temporary fixes) by developers and explicitly recorded in source code comments. In their work, they manually classified more than 100k comments and summarized 62 patterns that indicated SATD. By leveraging these patterns, a number of researchers further investigated different properties of SATD. For example, Maldonado and Shihab (2015) found that self-admitted technical debt can be classified into five main types - design debt, defect debt, documentation debt, requirement debt and test debt. Bavota and Russo (2016) conducted a large-scale empirical study on SATD and found that 25% of the SATD automatically identified by using the 62 patterns are likely to represent false positives. However, they didn't investigate how many true SATD comment are not identified by patterns (i.e., false negatives). Wehaibi et al. (2016) examined the relationship between SATD and software quality. They highlighted that although technical debt may have negative effects, its impact is not only related to defects, rather making the system more difficult to change in the future.

Inspired by the prior work, we also leverage source code comments to detect SATD. However, our approach is based on text mining, which is different from theirs. Besides, our approach achieves better performance, especially in terms of recall, which can help detect more SATD that are mistakenly ignored by Potdar and Shihab's pattern-based approach.

Most recently, Maldonado et al. (2017) proposed an approach to automatically identify SATD in a finer granularity (i.e., what type of SATD it is). Specifically, their approach would identify a comment as SATD on design, SATD on requirement or non-SATD. The problem studied in our work is different from theirs; we focus on identifying whether a comment contains SATD, which also includes other types of SATD (i.e., defect debt, documentation debt and test debt). Our approach is also different from theirs; they built a maximum entropy classifier based on natural language processing (NLP), and they didn't use feature selection or ensemble learning. However, the NLP-based approach can also be applied to our problem, and we compared its performance with ours in RQ1. The experiment results show that our approach outperforms the NLP-based approach, with an average improvement of 27.95% in terms of F1-score. Finally, compared with Maldonado et al.'s work, we perform empirical study on SATD comments in different perspectives and present different findings. Thus, we believe our work could be a complement to Maldonado et al.'s work.

7.2 Source Code Comments

A number of previous work on source code comments focused on investigating the relationship between comments and code. For example, Tan et al. proposed a tool called iComment to automatically analyze comments written in natural language to extract implicit program rules, and then they use these rules to automatically detect inconsistencies between comments and source code, indicating either bugs or bad comments (Tan et al. 2007). In their follow-on work, they studied the inconsistencies between Javadoc comments and method bodies (Tan et al. 2012). Malik et al. presented a large empirical study to better understand the rationale for updating comments, and they used the Random Forests algorithm to accurately predict the likelihood of a comment being updated (Malik et al. 2008). Fluri et al. proposed an approach to map code and comments to observe their co-evolution over multiple versions (Fluri et al. 2007).

Other work focused on using comments to assist in software development and maintenance. For example, Khamis proposed an automatic approach for assessing the quality of inline documentation using a set of heuristics, targeting both quality of language and consistency between source code and its comments (Khamis et al. 2010). Padioleau et al. studied 1,050 comments randomly sampled from Linux, FreeBSD, and OpenSolaris and found that 52.6% of these comments could be leveraged for improving reliability (Padioleau et al. 2009). Storey et al. investigated how task annotations can be used to support a variety of activities fundamental to articulation work within software development. They found that the use of task annotations varies from individuals to teams and if incorrectly managed, could negatively impact the maintenance of a system (Storey et al. 2008).

Our work is different from these prior work, since we focus on identifying SATD through source code comments. Although Potdar and Shihab (2014) also used source code comments to detect SATD, our approach is based on text mining, which is different from their pattern-based approach.

7.3 Text Mining in Software Engineering

There have been a number of studies on text mining in different areas of Software Engineering (Sun et al. 2010; Sun et al. 2011; Nguyen et al. 2012; Marcus and Maletic 2003; Haiduc et al. 2010; Zhou et al. 2012; Tian et al. 2012; Yang et al. 2017; Xia et al. 2017; Xia et al. 2015c, d; Yang et al. 2016; Zhang et al. 2016). For example, Sun et al. applied text mining and extend the similarity formula "BM25F" to accurately detect duplicated bug reports (Sun et al. 2010; Sun et al. 2011). Nguyen et al. further improved the performance by combining topic model with BM25F (Nguyen et al. 2012). Marcus and Maletic used Latent Semantic Indexing (LSI) to recover traceability links from documentation to source code (Marcus and Maletic 2003). Haiduc et al. applied automated text summarization technology to automatically produce simple textual descriptions of source code entities that developers can grasp easily, while capturing the code semantics precisely (Haiduc et al. 2010). Zhou et al. proposed BugLocator to find source code files that are likely to be relevant to a given bug report (Zhou et al. 2012). BugLocator ranks all files based on the textual similarity between the initial bug report and the source code using a revised Vector Space Model (rVSM), taking into consideration information about similar bugs that have been fixed before. Tian et al. proposed a semi-supervised learning algorithm to identify Linux bug fixing patches based on the changes and commit messages recorded in code repositories (Tian et al. 2012).

While we are motivated by these prior work, our work is still different from them since we apply text mining on source code comments to detect SATD.

8 Conclusion and Future Work

In this paper, we proposed an automated approach to detect self-admitted technical debt in source code comments. We use comments from a number of different source projects as training data to predict the label of the comment in a new target project. In our approach, we first preprocess the text description of all comments and apply feature selection on each source project to select top 10% features with the highest feature selection scores. Then we use the selected features to train a sub-classifier on each source project. After that, a composite classifier is build from these sub-classifiers and it will predict the label of a new comment in target project. The output of the composite classifier is the label predicted most often by its sub-classifiers, with ties broken arbitrarily.

To evaluate the performance of our approach, we first collected a dataset of source code comments from 8 open source projects which have a large number of comments and belong to different application domains. The experiment results show that, on every target project, our approach achieves the best performance in terms of F1-score. The F1-score achieved by our approach ranges between 0.518 - 0.841, with an average of 0.737, which improves significantly over the baseline methods. On average, our approach improves the F1-scores over Potdar and Shihab's approach, NBM baseline, SVM baseline and kNN baseline by 499.19%, 58.49%, 882.67% and 205.81%, respectively.

In the future, we plan to evaluate our approach on datasets from more software projects. To further improve the performance our approach, we plan to apply deep learning techniques such as word embedding (Mikolov et al. 2013). We also plan to develop an automated tool to make the development team being aware of the SATD when it is introduced. Finally, We plan to contact some contributors in open source projects to ask them about their views on SATD comments.

Acknowledgements The authors thank to all the developers who participated in this study. This research was supported by NSFC Program (No. 61602403 and 61572426), and National Key Technology R&D Program of the Ministry of Science and Technology of China (No. 2015BAH17F01).

References

Arisholm E, Briand LC, Fuglerud M (2007) Data mining techniques for building fault-proneness models in telecom java software. In: The 18th IEEE international symposium on software reliability (ISSRE'07), IEEE, pp 215–224

Bavota G, Russo B (2016) A large-scale empirical study on self-admitted technical debt. In: Proceedings of the 13th international conference on mining software repositories, MSR '16, pp 315–326

Breiman L (1996) Bagging predictors. Mach Learn 24(2):123–140

Brown N, Cai Y, Guo Y, Kazman R, Kim M, Kruchten P, Lim E, MacCormack A, Nord R, Ozkaya I et al (2010) Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP workshop on Future of software engineering research. ACM, pp 47–52

Cohen J (1968) Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit. Psychol Bull 70(4):213

Cunningham W (1993) The wycash portfolio management system. ACM SIGPLAN OOPS Messenger 4(2):29–30

Fluri B, Wursch M, Gall HC (2007) Do code and comments co-evolve? On the relation between source code and comment changes. In: 14th working conference on reverse engineering (WCRE 2007). IEEE, pp 70–79

Guo Y, Seaman C, Gomes R, Cavalcanti A, Tonin G, Da Silva FQ, Santos AL, Siebra C (2011) Tracking technical debtan exploratory case study. In: 2011 27th IEEE international conference on software maintenance (ICSM). IEEE, pp 528–531

Haiduc S, Aponte J, Marcus A (2010) Supporting program comprehension with source code summarization. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering-volume 2. ACM, pp 223–226

Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The weka data mining software: an update. ACM SIGKDD Explorations Newsletter 11(1):10–18

Hall MA (1999) Correlation-based feature selection for machine learning. PhD thesis, The University of Waikato

Han J, Kamber M, Pei J (2006) Data mining: concepts and techniques. Morgan Kaufmann

He H, Garcia EA (2009) Learning from imbalanced data. IEEE Trans Knowl Data Eng 21(9):1263–1284

Jiang T, Tan L, Kim S (2013) Personalized defect prediction. In: 2013 IEEE/ACM 28th international conference on automated software engineering (ASE). IEEE, pp 279–289

Khamis N, Witte R, Rilling J (2010) Automatic quality assessment of source code comments: the javadocminer. In: International conference on application of natural language to information systems. Springer, pp 68–79

Kruchten P, Nord RL, Ozkaya I, Falessi D (2013) Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt. ACM SIGSOFT Softw Eng Notes 38(5):51–54

Lim E, Taksande N, Seaman C (2012) A balancing act: what software practitioners have to say about technical debt. Softw IEEE 29(6):22–27

Maldonado E, Shihab E (2015) Detecting and quantifying different types of self-admitted technical debt. In: Proceedings of the 7th IEEE international workshop on managing technical debt (MTD'15), pp 9–15

Maldonado E, Shihab E, Tsantalis N (2017) Using natural language processing to automatically detect self-admitted technical debt. IEEE Transactions on Software Engineering

Malik H, Chowdhury I, Tsou HM, Jiang ZM, Hassan AE (2008) Understanding the rationale for updating a functions comment. In: IEEE international conference on software maintenance, 2008. ICSM 2008. IEEE, pp 167–176

Marcus A, Maletic JI (2003) Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings of the 25th international conference on software engineering, 2003. IEEE, pp 125–135

Marinescu R (2004) Detection strategies: Metrics-based rules for detecting design flaws. In: Proceedings of the 20th IEEE international conference on software maintenance, 2004. IEEE, pp 350–359

Marinescu R, Ganea G, Verebi I (2010) incode: Continuous quality assessment and improvement. In: 2010 14th European conference on software maintenance and reengineering (CSMR). IEEE, pp 274–275

McCallum A, Nigam K et al (1998) A comparison of event models for naive bayes text classification. In: AAAI-98 Workshop on learning for text categorization. Citeseer, vol 752, pp 41–48

Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J (2013) Distributed representations of words and phrases and their compositionality. In: Advances in neural information processing systems, pp 3111–3119

Nguyen AT, Nguyen TT, Nguyen TN, Lo D, Sun C (2012) Duplicate bug report detection with a combination of information retrieval and topic modeling. In: Proceedings of the 27th IEEE/ACM international conference on automated software engineering (ASE), 2012. IEEE, pp 70–79

Padioleau Y, Tan L, Zhou Y (2009) Listening to programmers taxonomies and characteristics of comments in operating system code. In: Proceedings of the 31st international conference on software engineering, IEEE computer society, pp 331–341

Potdar A, Shihab E (2014) An exploratory study on self-admitted technical debt. In: 2014 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 91–100

Rahman F, Posnett D, Devanbu P (2012) Recalling the imprecision of cross-project defect prediction. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering. ACM, p 61

Salton G, Wong A, Yang CS (1975) A vector space model for automatic indexing. Commun ACM 18(11):613–620

Seaman C, Guo Y, Izurieta C, Cai Y, Zazworka N, Shull F, Vetrò A (2012) Using technical debt data in decision making: Potential decision approaches. In: Proceedings of the 3rd international workshop on managing technical debt. IEEE Press, pp 45–48

Sebastiani F (2002) Machine learning in automated text categorization. ACM Comput Surv (CSUR) 34(1):1–47

Shihab E, Ihara A, Kamei Y, Ibrahim WM, Ohira M, Adams B, Hassan AE, Ki M (2013) Studying re-opened bugs in open source software. Empir Softw Eng 18(5):1005–1042

Storey MA, Ryall J, Bull RI, Myers D, Singer J (2008) Todo or to bug. In: 2008 ACM/IEEE 30th international conference on software engineering. ICSE'08. IEEE, pp 251–260

Sun C, Lo D, Wang X, Jiang J, Khoo SC (2010) A discriminative model approach for accurate duplicate bug report retrieval. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering. ACM, vol 1, pp 45–54

Sun C, Lo D, Khoo SC, Jiang J (2011) Towards more accurate retrieval of duplicate bug reports. In: Proceedings of the 2011 26th IEEE/ACM international conference on automated software engineering. IEEE Computer Society, pp 253–262

Tan L, Yuan D, Krishna G, Zhou Y (2007) /* icomment: Bugs or bad comments?*. In: ACM SIGOPS operating systems review. ACM, vol 41, pp 145–158

Tan SH, Marinov D, Tan L, Leavens GT (2012) @ Tcomment: Testing javadoc comments to detect comment-code inconsistencies. In: 2012 IEEE Fifth international conference on software testing, Verification and Validation, IEEE, pp 260–269

Tian Y, Lawall J, Lo D (2012) Identifying linux bug fixing patches. In: 2012 34th international conference on software engineering (ICSE). IEEE, pp 386–396

Valdivia Garcia H, Shihab E (2014) Characterizing and predicting blocking bugs in open source projects. In: Proceedings of the 11th working conference on mining software repositories. ACM, pp 72–81

Vassallo C, Zampetti F, Romano D, Beller M, Panichella A, Penta MD, Zaidman A (2016) Continuous delivery practices in a large financial organization. In: Proceedings of the international conference on software maintenance and evolution (ICSME), ICSME '16, p To Appear

Wehaibi S, Shihab E, Guerrouj L (2016) Examining the impact of self-admitted technical debt on software quality. In: Proceedings of the 23rd IEEE international conference on software analysis, evolution, and reengineering (SANER'16)

Wilcoxon F (1945) Individual comparisons by ranking methods. Biom Bull 1(6):80–83

Xia X, Lo D, Wang X, Zhou B (2013) Tag recommendation in software information sites. In: Proceedings of the 10th working conference on mining software repositories. IEEE Press, pp 287–296

Xia X, Lo D, Qiu W, Wang X, Zhou B (2014) Automated configuration bug report prediction using text mining. In: 2014 IEEE 38th annual computer software and applications conference (COMPSAC). IEEE, pp 107–116

Xia X, Lo D, Shihab E, Wang X, Yang X (2015a) Elblocker: Predicting blocking bugs with ensemble imbalance learning. Inf Softw Technol 61:93–106

Xia X, Lo D, Shihab E, Wang X, Zhou B (2015b) Automatic, high accuracy prediction of reopened bugs. Autom Softw Eng 22(1):75–109

Xia X, Lo D, Wang X, Yang X (2015c) Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In: 2015 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 261–270

Xia X, Lo D, Wang X, Zhou B (2015d) Dual analysis for recommending developers to resolve bugs. J Softw Evol Process 27(3):195–220

Xia X, Lo D, Pan SJ, Nagappan N, Wang X (2016a) Hydra: Massively compositional model for cross-project defect prediction. IEEE Trans Softw Eng 42(10):977–998

Xia X, Lo D, Wang X, Yang X (2016b) Collective personalized change classification with multiobjective search. IEEE Trans Reliab 65(4):1810–1829

Xia X, Shihab E, Kamei Y, Lo D, Wang X (2016c) Predicting crashing releases of mobile applications. In: Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement. ACM, p 29

Xia X, Lo D, Ding Y, Al-Kofahi JM, Nguyen TN, Wang X (2017) Improving automated bug triaging with specialized topic model. IEEE Trans Softw Eng 43(3):272–297

Xu B, Ye D, Xing Z, Xia X, Chen G, Li S (2016) Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering. ACM, pp 51–62

Yang X, Lo D, Xia X, Bao L, Sun J (2016) Combining word embedding with information retrieval to recommend similar bug reports. In: 2016 IEEE 27th international symposium on software reliability engineering (ISSRE). IEEE, pp 127–137

Yang XL, Lo D, Xia X, Huang Q, Sun JL (2017) High-impact bug report identification with imbalanced learning strategies. J Comput Sci Technol 32:1

Yang Y, Pedersen JO (1997) A comparative study on feature selection in text categorization. In: ICML, vol 97, pp 412–420

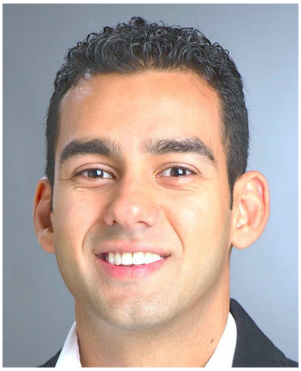
Zazworka N, Shaw MA, Shull F, Seaman C (2011) Investigating the impact of design debt on software quality. In: Proceedings of the 2nd workshop on managing technical debt. ACM, pp 17–23

Zazworka N, Spínola RO, Vetro A, Shull F, Seaman C (2013) A case study on effectively identifying technical debt. In: Proceedings of the 17th international conference on evaluation and assessment in software engineering. ACM, pp 42–47

Zhang Y, Lo D, Xia X, Le TDB, Scanniello G, Sun J (2016) Inferring links between concerns and methods with multi-abstraction vector space model. In: 2016 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 110–121

Zhou J, Zhang H, Lo D (2012) Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In: 2012 34th international conference on software engineering (ICSE). IEEE, pp 14–24

Qiao Huang is currently a Ph.D. student in the College of Computer Science and Technology, Zhejiang University, China. His research interests include mining software repositories and empirical software engineering.



Emad Shihab is an Assistant Professor in the Department of Computer Science and Software Engineering at Concordia University. He received his PhD from Queens University. Dr. Shihab's research interests are in Software Quality Assurance, Mining Software Repositories, Technical Debt, Mobile Applications and Software Architecture. He worked as a software research intern at Research in Motion in Waterloo, Ontario and Microsoft Research in Redmond, Washington. Dr. Shihab is a member of the IEEE and ACM. More information can be found at http://das.encs.concordia.ca.

Xin Xia received his PhD degree in computer science from the College of Computer Science and Technology, Zhejiang University, China in 2014. He is currently a post-doc research fellow in the software practices lab at the University of British Columbia, Canada. His research interests include software analytic, empirical study, and mining software repository.

David Lo received his PhD degree from the School of Computing, National University of Singapore in 2008. He is currently an Associate Professor in the School of Information Systems, Singapore Management University. He has close to 10 years of experience in software engineering and data mining research and has more than 200 publications in these areas. He received the Lee Foundation Fellow for Research Excellence from the Singapore Management University in 2009, and a number of international research awards including several ACM distinguished paper awards for his work on software analytics. He has served as general and program co-chair of several prestigious international conferences (e.g., IEEE/ACM International Conference on Automated Software Engineering), and editorial board member of a number of high-quality journals (e.g., Empirical Software Engineering).

Shanping Li received his Ph.D. degree from the College of Computer Science and Technology, Zhejiang University in 1993. He is currently a professor in the College of Computer Science and Technology, Zhejiang University. His research interests include Software Engineering, Distributed Computing, and the Linux Operating System.